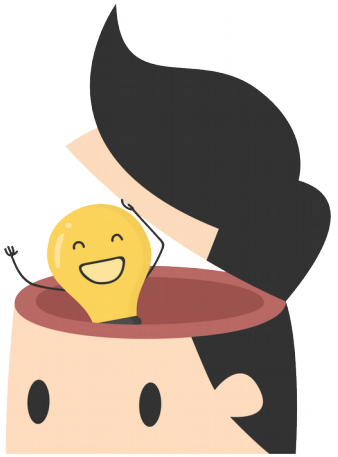
Python – Class IX



Functions

LavaTech Technology

What is a function in Python?

- In Python, function is a group of related statements that perform a specific task.

- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

- Furthermore, it avoids repetition and makes code reusable.

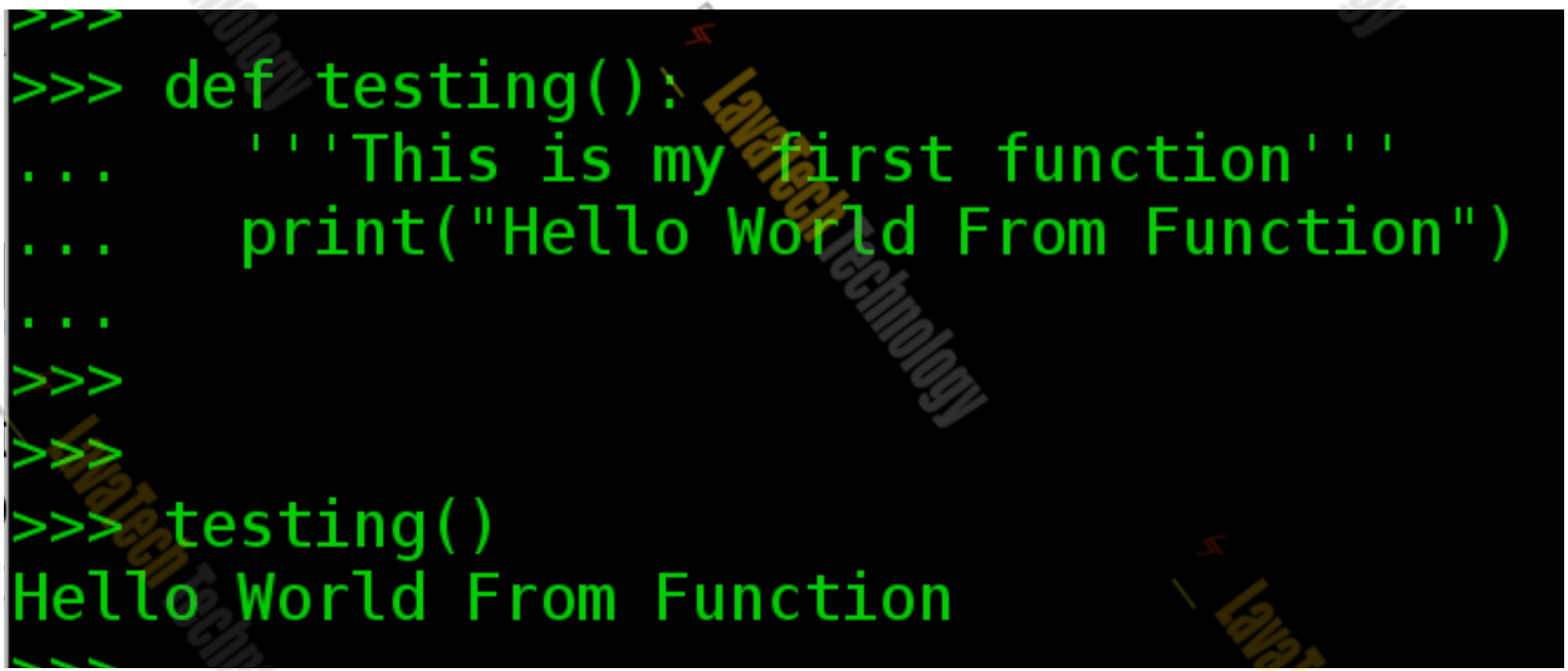
Syntax of Function -

```
def function_name(parameters):
    """docstring"""
    statement(s)
```

- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of function header.
- Optional documentation string (docstring) to describe what the function does.

How to call a function in python?

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.



Docstring

- The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does.
- Although optional, documentation is a good programming practice.
- In the recent example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as __doc__ attribute of the function.

```
>>>
>>> def testing():
...     '''This is my first function'''
...     print("Hello World From Function")
...
>>>
>>>
>>> testing()
Hello World From Function
>>>
>>> print(testing.__doc__)
This is my first function
```

The return statement

- The return statement is used to exit a function and go back to the place from where it was called.

  Syntax of return

  return [expression_list]

- This statement can contain expression which gets evaluated and the value is returned.
- If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

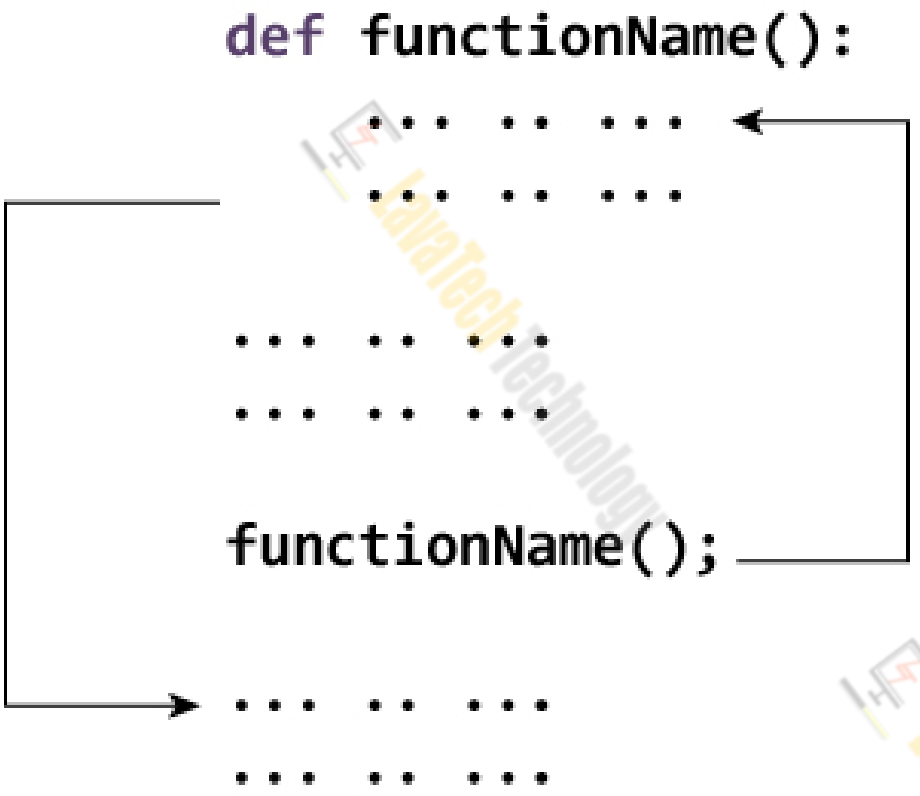
```
>>> def testing():
...     print("Hello World From Python")
...
>>>
>>> print(testing())
Hello World From Python
None
>>>
```

Example

```
>>>
>>> def absolute_value(num):
...     '''This functions uses return statement'''
...     if num >= 0:
...         return num
...     else:
...         return -num
...
>>>
>>> print("The absolute value of -2 is: ", absolute_value(-2))

The absolute value of -2 is:  2
>>>
>>> print("The absolute value of 10 is: ", absolute_value(-10))
The absolute value of 10 is:  10
>>>
```

How Function works in Python?



Scope and Lifetime of variables

- Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a **local scope.**

- Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

```
>>>
>>> def test():
...     x=10
...     print("Value of x is: ",x)
...
>>>
>>> x=50
>>> test()
Value of x is:  10
>>>
>>> print("Value is x is: ",x)
Value is x is:  50
```

On the other hand, variables outside of the function are visible from inside. They have a global scope.

```
>>> a="jack"
>>>
>>> def test():
...     print("Value of a is: ", a)
...
>>>
>>> print(test())
Value of a is:  jack
None
```

We can read these values from inside the function but cannot change (write) them.

```
>>> a="jack"
>>>
>>> def test():
...     print("Value of a is: ", a)
...     a="ram"
...     print("Value of a is: ", a)
...
>>>
>>> a="Jill"
>>>
>>> print("Value of a is: ", a)
Value of a is:  Jill
>>>
>>> print(test())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in test
UnboundLocalError: local variable 'a' referenced before assignment
```

In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

```
>>> x=10
>>>
>>> def f1():
...    global x
...    x=x+10
...    print("Modified value of x is: ",x)
...
>>>
>>> print("Value of x is: ",x)
Value of x is:  10
>>>
>>> f1()
Modified value of x is:  20
>>>
>>> print("Value of x is: ",x)
Value of x is:  20
>>>
```

Types of Functions

Basically, we can divide functions into the following two types:

Built-in functions - Functions that are built into Python.
User-defined functions - Functions defined by the users themselves.

Python Built-in Function

- The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions.
- For example, print() function prints the given object to the standard output device (screen) or to the text stream file.
- In Python 3.6 (latest version), there are 68 built-in functions. Below link displays all the built-in functions - https://docs.python.org/3.6/library/functions.html

Python Function Arguments

We have seen so far about defining a function and calling it with arguments. Otherwise, the function call will result into an error. Here is an example.

```
>>>
>>> var1="Python Class"
>>>
>>> def test(var1,var2,var3):
...   print(var1,var2,var3)
...
>>>
>>> var2="Functions Topic"
>>>
>>> var3="Covered"
>>>
>>> test(var2,var3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: test() missing 1 required positional argument: 'var3'
>>>
>>> test(var1,var2,var3)
Python Class Functions Topic Covered
```

Variable Function Arguments

- Up until now functions had fixed number of arguments. In Python there are other ways to define a function which can take variable number of arguments.Three different forms of this type are described below.
- **Python Default Arguments**
  - Function arguments can have default values in Python.
  - We can provide a default value to an argument by using the assignment operator (=). Here is an example.

```
>>> name="Kumar"
>>>
>>> msg="hi, wats up"
>>>
>>> def greet(name,msg="GM"):
...     print(name,msg)
...
>>>
>>> greet(name)
Kumar GM
>>>
>>> greet(name,msg)
Kumar hi, wats up
>>>
```

Non-default arguments cannot follow default arguments.

```
>>> def test(name="Raj",msg="hi",msg2):
...   print(name,msg,msg2)
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
>>>
```

- We can call the function by passing argument in different order, which means the order (position) of the arguments can be changed.
- Following calls to the above function are all valid and produce the same result.

```
>>> def greet(name,msg="Hi, wats up!"):
...     print(name,msg)
...
>>>
>>> greet(name = "Bruce")
Bruce Hi, wats up!
>>>
>>> greet(name = "Bruce" , msg = "Bye")
Bruce Bye
>>>
>>> greet(msg = "Bye" , name = "Raj")
Raj Bye
>>>
>>> greet(msg = "Okay let's see" , name = "Raj")
Raj Okay let's see
>>>
>>> greet("Okay let's see" , name = "Raj")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: greet() got multiple values for argument 'name'
>>>
```

Python Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function.

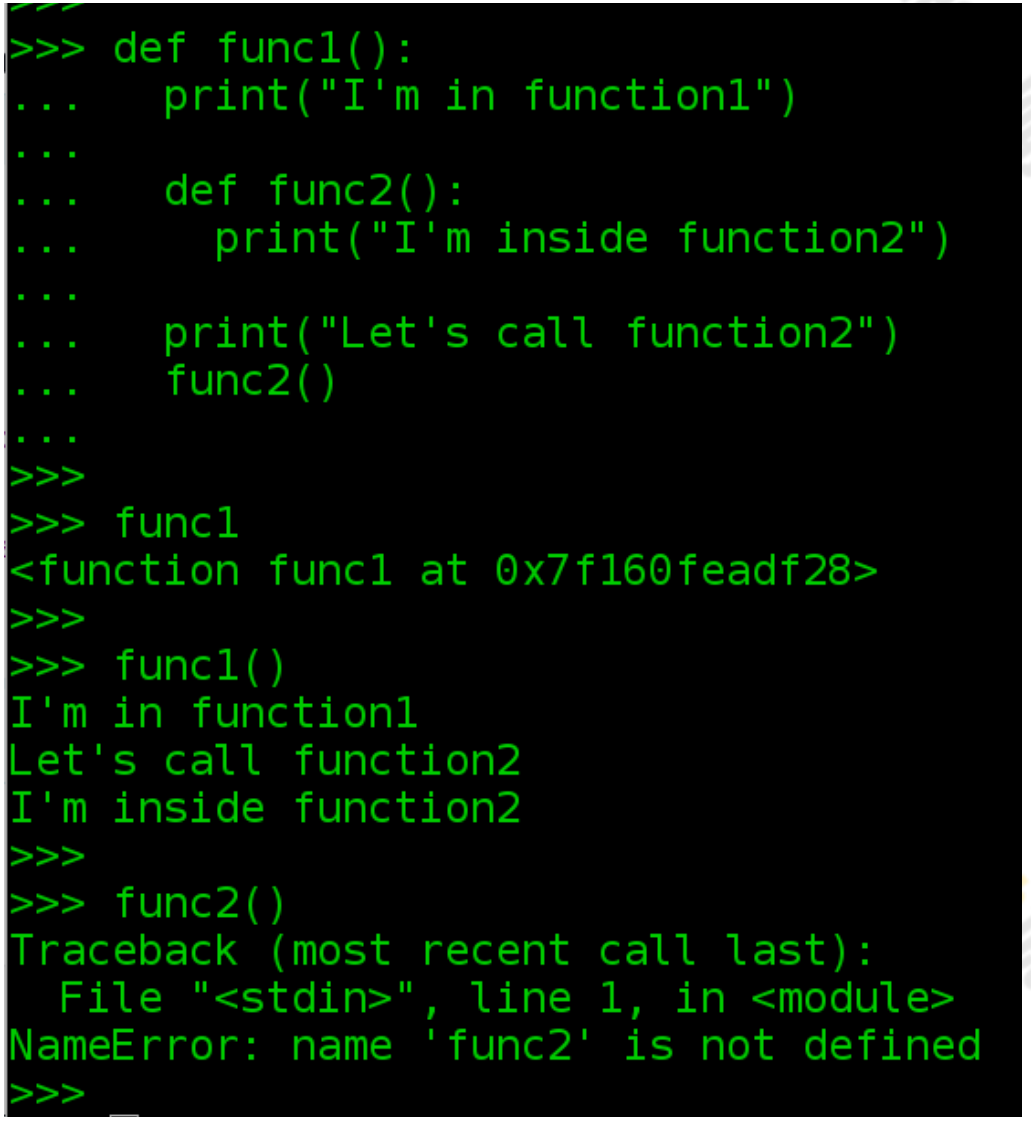
Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.

In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument.

```
>>>
>>> def greet(*names):
...     print(type(names))
...
...     for x in names:
...         print("GM", x)
...
>>>
>>>
>>> greet("Raj","Ram","Ravi","Kavi")
<class 'tuple'>
GM Raj
GM Ram
GM Ravi
GM Kavi
>>>
```

Nested Functions

- A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.



Python Closures

- The criteria that must be met to create closure in Python are summarized in the following points.
  - We must have a nested function (function inside a function).
  - The nested function must refer to a value defined in the enclosing function.
  - The enclosing function must return the nested function.

```
>>> def f1():
...   def f2():
...     print("This is my Secret Function")
...   return f2
...
>>>
>>> var=f1()
>>>
>>> var
<function f1.<locals>.f2 at 0x7f160e184bf8>
>>>
>>> var()
This is my Secret Function
>>>
>>> del f1
>>>
>>> f1()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'f1' is not defined
>>>
>>> var()
This is my Secret Function
>>>
```

Global Keyword

- In Python, global keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.
- The basic rules for global keyword in Python are:
  - When we create a variable inside a function, it’s local by default.
  - When we define a variable outside of a function, it’s global by default. You don’t have to use global keyword.
  - We use global keyword to read and write a global variable inside a function.
  - Use of global keyword outside a function has no effect

Accessing global Variable From Inside a Function

```
>>> name="Jill"
>>>
>>> def test():
...     print("Value of name is:", name)
...
>>>
>>> test()
Value of name is: Jill
```

Modifying Global Variable From Inside the Function

Below error is faced on trying to modify the global values from inside a function.

```
>>> name="Jill"
>>>
>>> def test():
...     name=name+" "+"Jack"
...     print(name)
...
>>>
>>> test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in test
UnboundLocalError: local variable 'name' referenced before assignme
nt
>>>
```

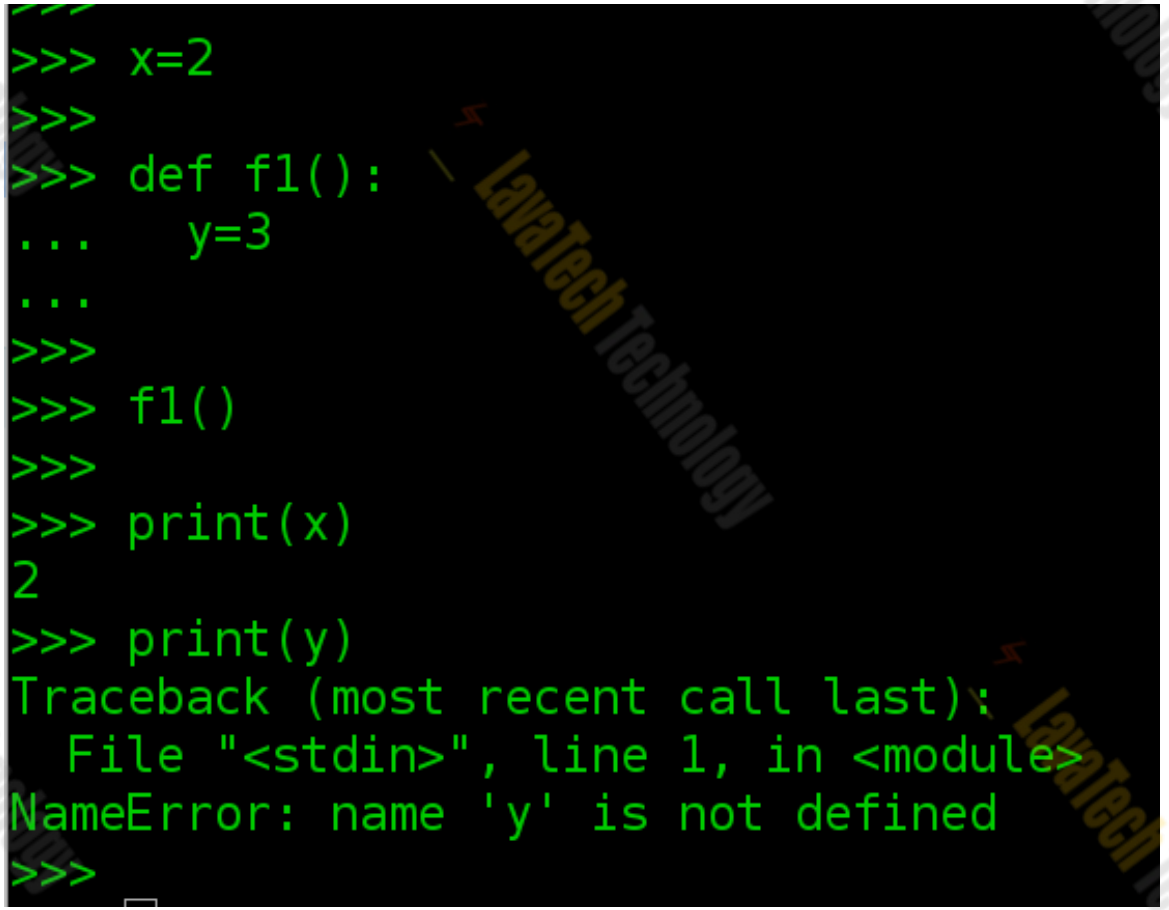
Changing Global Variable From Inside a Function using global

- We can only access the global variable but cannot modify it from inside the function.
- The solution for this is to use the global keyword.

```
>>> x=10
>>>
>>> def f1():
...     global x
...     x=x+10
...     print("Modified value of x is: ",x)
...
>>>
>>> print("Value of x is: ",x)
Value of x is:  10
>>>
>>> f1()
Modified value of x is:  20
>>>
>>> print("Value of x is: ",x)
Value of x is:  20
```

Local Variables

- A variable declared inside the function's body or in the local scope is known as local variable.



Global variable and Local variable with same name

```
>>>
>>> x=40
>>>
>>> def f1():
...     x=70
...     print(x)
...
>>>
>>> print(x)
40
>>> f1()
70
>>>
```

Nonlocal Variables

- Nonlocal variable are used in nested function whose local scope is not defined. This means, the variable can be neither in the local nor the global scope.
- Let's see an example on how a global variable is created in Python.
- We use nonlocal keyword to create nonlocal variable.

```
>>>
>>> def f1():
...     y=20
...     def f2():
...         nonlocal y
...         print("Value of y is: ",y)
...         y=30
...         print("Value of y is: ",y)
...     f2()
...     print("Value of y in f1() is: ",y)
...
>>>
>>> f1()
Value of y is:  20
Value of y is:  30
Value of y in f1() is:  30
>>>
```

Note: nonlocal variable should be the one that's defined by outer function

```
>>> def f1():
...     y=50
...     def f2():
...         nonlocal z
...         z=60
...         print("Value of z is: ",z)
...
  File "<stdin>", line 4
SyntaxError: no binding for nonlocal 'z' found
```

Global,local and nonlocal scope

```
root@server:~# cat gobal_local_nonlocal.py
a = 0    #1. global variable with respect to every function in program

def f():
    a = 0          #2. nonlocal with respect to function g
    def g():
        nonlocal a
        a=a+1
        print("The value of 'a' using nonlocal is ", a)
    def h():
        global a               #3. using global variable
        a=a+5
        print("The value of a using global is ", a)
    def i():
        a = 0              #4. variable separated from all others
        print("The value of 'a' inside a function is ", a)

    g()
    h()
    i()
print("The value of 'a' global before any function", a)
f()
print("The value of 'a' global after using function f ", a)
root@server:~#
root@server:~#
root@server:~# python3 gobal_local_nonlocal.py
The value of 'a' global before any function 0
The value of 'a' using nonlocal is  1
The value of a using global is  5
The value of 'a' inside a function is  0
The value of 'a' global after using function f  5
root@server:~#
```

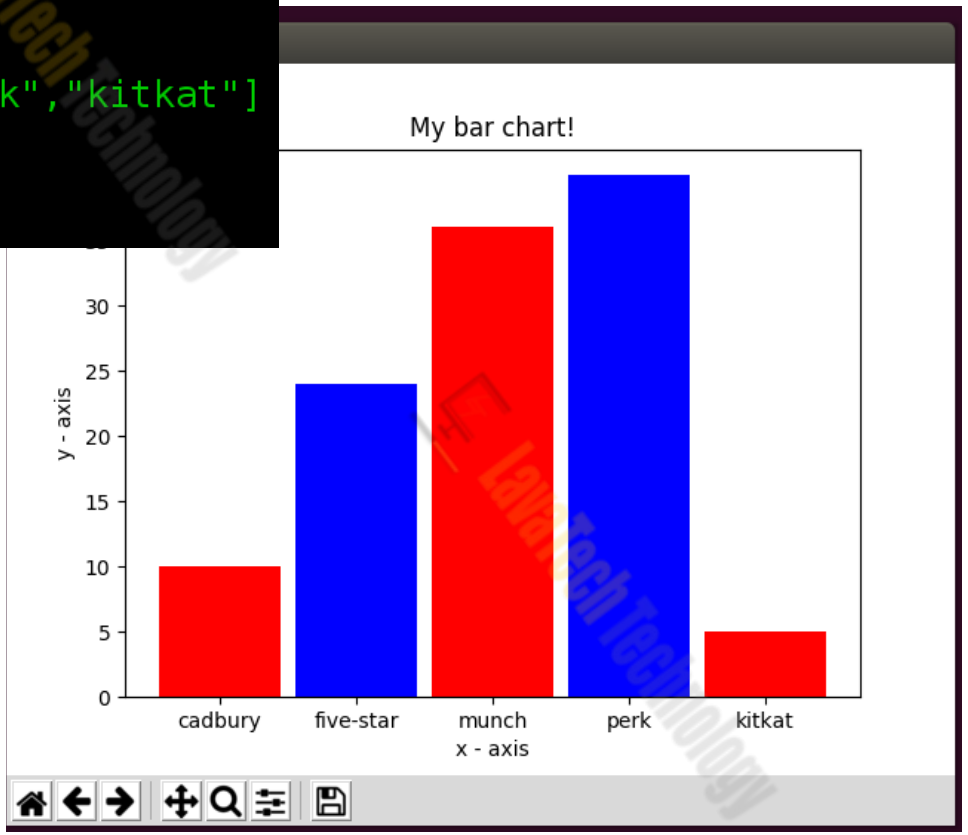
Displaying graph using functions

```
root@server:~# cat g2.py
import matplotlib.pyplot as plt

def graph(items,sales):
    tot_sales=sum(sales)
    plt.bar(items,sales,width=0.9,color=['red','blue'])
    plt.xlabel('x - axis')
    plt.ylabel('y - axis')
    plt.title('My bar chart!')
    plt.show()
    return tot_sales

items = ["cadbury", "five-star","munch","perk","kitkat"]
sales = [10, 24, 36, 40, 5]
tot_sales=graph(items,sales)
print("Total Sales is: ",tot_sales)
```

```
root@server:~# python3 g2.py
Total Sales is:  115
root@server:~#
```



What is Python lambda?

- Before trying to understand what a Python lambda is, let's first try to understand what a Python function is at a much deeper level.
- As you already know, everything in Python is an object.
- For example, when we run this simple line of code.
- x = 5
- What actually happens is we're creating a Python object of type int that stores the value 5.
- x is essentially a symbol that is referring to that object.
- Now let's check the type of x and the address it is referring to.
- We can easily do that using the type and the id built-in functions.

```
>>> type(x)
<class 'int'>
>>> id(x)
4308964832
```

- Now what happens when we define a function like this one:

```
>>> def f(x):
...     return x * x
...
```

- Let’s repeat the same exercise from above and inspect the type of “f” and its id.

```
>>> def f(x):
...     return x * x
...
>>> type(f)
<class 'function'>
>>> id(f)
4316798080
```

- So it turns out there is a function class in Python and the function f that we just defined is an instance of that class.
- Exactly like how x was an instance of the integer class.
- In other words, you can literally think about functions the same way you think about variables.
- The only difference is that a variable stores data whereas a function stores code.
- That also means you can pass functions as arguments to other functions, or even have a function be the return value of another function.
- Let's look at a simple example where you can pass the above function f to another function.

```
def f(x):
    return x * x

def modify_list(L, fn):
    for idx, v in enumerate(L):
        L[idx] = fn(v)

L = [1, 3, 2]
modify_list(L, f)
print(L)

#output: [1, 9, 4]
```

Anonymous Function

- In Python, anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.
- Hence, anonymous functions are also called lambda functions.

How to use lambda Functions in Python?

- A lambda function in python has the following syntax.
- Syntax of Lambda Function in python -

lambda arguments: expression

Example -

```
>>> f = lambda x: x * x
>>> type(f)
<class 'function'>
```

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming besides other types of expressions in functions.
- Let's look at this example and try to understand the difference between a normal def defined function and lambda function. This is a program that returns the cube of a given value:

```
>>> def cube(y):
...     return y*y*y
...
>>>
>>> g = lambda x: x*x*x
>>>
>>> print(g(7))
343
>>>
>>> print(cube(5))
125
>>>
```

- **Without using Lambda :**
  - Here, both of them returns the cube of a given number.
  - But, while using def, we needed to define a function with a name cube and needed to pass a value to it.
  - After execution, we also needed to return the result from where the function was called using the return keyword.
- **Using Lambda :**
  - Lambda definition does not include a “return” statement, it always contains an expression which is returned.
  - We can also put a lambda definition anywhere a function is expected, and we don’t have to assign it to a variable at all.
  - This is the simplicity of lambda functions.

- Examples -
- A lambda function that adds 10 to the number passed in as an argument, and print the result:

```
x = lambda a : a + 10

print(x(5))
```

- A lambda function that multiplies argument a with argument b and print the result:

```
x = lambda a, b : a * b

print(x(5, 6))
```

- A lambda function that sums argument a, b, and c and print the result:

```
x = lambda a, b, c : a + b + c

print(x(5, 6, 2))
```

Lambdas with no arguments

- Say you want to define a lambda function that takes no arguments and returns True.
- You can achieve this with the following code.

```
>>> f = lambda: True
>>> f()
True
```

Multiline lambdas

- Yes, at some point in your life you will be wondering if you can have a lambda function with multiple lines.
- And the answer is: No you can't ☺
- Python lambda functions accept only one and only one expression.
- If your function has multiple expressions/statements, you are better off defining a function the traditional way instead of using lambdas.

Why Use Lambda Functions?

- The power of lambda is better shown when you use them as an anonymous function inside another function.
- Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

- Use that function definition to make a function that always doubles the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n
```

- mydoubler = myfunc(2)
- print(mydoubler(11))

- Or, use the same function definition to make a function that always triples the number you send in:
- Example

```
def myfunc(n):
  return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

- Or, use the same function definition to make both functions, in the same program:
- Example

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

mytripler = myfunc(3)

print(mydoubler(11))

print(mytripler(11))
```

Lambda function in List Comprehensions

- When using lambda functions with list comprehensions, one must use the first method shown below and not second one -
- Mind you, both type(f) and type(lambda x: x*x) return the same type.
- The first one create a single lambda function and calls it ten times.
- The second one doesn't call the function. It creates 10 different lambda functions. It puts all of those in a list.
- First Method -

```
f = lambda x: x*x
l1=[f(x) for x in range(10)]
print(l1)
```

- Second Method -

```
[lambda x: x*x for x in range(10)]
```

Lambda inside a Python loop

- You can create a list of lambdas in a python loop using the following syntax:

```
def square(x): return lambda : x*x

listOfLambdas = [square(i) for i in [1,2,3,4,5]]

for f in listOfLambdas: print f()
```

- This will give the output:

```
1
4
9
16
25
```

Lambda Function using ‘if’ statement

- Use if..else one linear syntax when using if inside lambda functions -

```
x= lambda a,b,c: a if a>b and a>c else b if b>c else c
```

Map, Filter and Reduce Functions

- Mostly lambda functions are passed as parameters to a function which expects a function objects as parameter like map, reduce, filter functions

map

- map functions expects a function object and any number of iterables like list, dictionary, etc. It executes the function_object for each element in the sequence and returns a list of the elements modified by the function object.
- map(function_object, iterable1, iterable2,…)

```
>>>
>>> def multiply(x):
...     return x * 2
...
>>>
>>> map(multiply,[1,2,3,4,5])
<map object at 0x7efe3f32a320>
>>>
>>> list(map(multiply,[1,2,3,4,5]))
[2, 4, 6, 8, 10]
>>>
```

- In the above example, map executes multiply2 function for each element in the list i.e. 1, 2, 3, 4 and returns [2, 4, 6, 8]
- Let’s see how we can write the above code using map and lambda.

```
>>>
>>> map(lambda x : x*2, [1, 2, 3, 4]) #Output [2, 4, 6, 8]
<map object at 0x7faff1574550>
>>>
>>> list(map(lambda x : x*2, [1, 2, 3, 4]))
[2, 4, 6, 8]
>>>
```

- Iterating over a dictionary using map and lambda
- In the below example, each dict of dict_a will be passed as parameter to the lambda function. Result of lambda function expression for each dict will be given as output.

```
>>>
>>> dict_a=[{"name":"Raj","points":10},{"name":"Ravi","points":20}]
>>>
>>> list(map(lambda x : x['name'], dict_a))
['Raj', 'Ravi']
>>>
>>> list(map(lambda x : x['points']*10, dict_a))
[100, 200]
>>>
>>> list(map(lambda x : x['name']=="Raj", dict_a))
[True, False]
>>>
```

- Multiple iterables to the map function
- We can pass multiple sequences to the map functions as shown below:

```
>>> list_a=[1,2,3]
>>>
>>> list_b=[10,20,30]
>>>
>>> list(map(lambda x,y: x+y , list_a , list_b ))
[11, 22, 33]
>>>
```

- In Python3, map function returns an iterator or map object which gets lazily evaluated. Just like zip function is lazily evaluated.
- Neither we can access the elements of the map object with index nor we can use len() to find the length of the map object
- Hence we have to use list() function.

filter

- Basic syntax
- filter(function_object, iterable)
- Like map function, filter function also returns a list of element. Unlike map function filter function can only have one iterable as input.
- Example:
- Even number using filter function

```
>>>
>>> a=[1,2,3,4,5,6]
>>>
>>> list(filter(lambda x:x%2 ==0 , a))
[2, 4, 6]
>>>
```

Filter list of dicts

```
>>>
>>> dict_a=[{"name":"Raj","points":10},{"name":"Ravi","points":20}]
>>>
>>> list(filter(lambda x: x['name'] == 'python', dict_a))
[]
>>>
>>> list(filter(lambda x: x['name'] == 'Raj', dict_a))
[{'name': 'Raj', 'points': 10}]
>>>
```

Reduce Function

- The reduce() function accepts a function and a sequence and returns a single value calculated as follows:
- Initially, the function is called with the first two items from the sequence and the result is returned.
- The function is then called again with the result obtained in step 1 and the next value in the sequence. This process keeps repeating until there are items in the sequence.
- The syntax of the reduce() function is as follows:

  Syntax: reduce(function, sequence[, initial]) -> value

- reduce() was a built-in function. However, in Python 3, it is moved to functools module. Therefore to use it, you have to first import it as follows:
- This reduce() call perform the following operation:
- (((1 + 2) + 3) + 4) => 10

Example

```
>>>
>>> l1=[100,200,300,400,500,600]
>>>
>>> import functools
>>>
>>> functools.reduce(lambda x,y: x+y, l1)
2100
>>>
>>> functools.reduce(lambda x,y: x*y, l1)
720000000000000
>>>
>>> functools.reduce(lambda x,y: x/y, l1)
1.3888888888888889e-11
>>>
>>> functools.reduce(lambda x,y: x-y, l1)
-1900
>>>
>>> functools.reduce(lambda x,y: x*3, l1)
24300
>>>
>>> functools.reduce(lambda x,y: x*2, l1)
3200
>>>
```